

What's the deal with dependent types?

Conor McBride

University of Nottingham
ctm@cs.nott.ac.uk

In this talk, I identify key ways in which dependently typed programming confounds our common expectations, and then transcends them. I expose language design issues where dependent types require a new deal. I hope to persuade you that this is not a big deal, and that it is a good deal. I discuss the impact of these issues on Epigram [3], the language designed by myself and James McKinna, and still evolving with help from Thorsten Altenkirch [1] and our colleagues in Nottingham and St Andrews.

It is standard to identify the syntax of a programming language, say λ -terms or pattern-matching equations, then equip that syntax with an operational semantics. We might then try to identify a system of types and type annotations which strikes a good balance between liberty, security and bureaucracy. What happens if we follow this approach, adopting a *dependent* type system?

We can start by writing down a λ -calculus with a suitable operational semantics and a dependent type system. I shall show you Epigram's: we have structured the syntax of terms specifically to clarify the flow of type information and reduce annotation.

Typechecking involves checking equality of normal forms of types, not just syntactic equality, so we need an operational semantics which preserves type information and computes under binders. This certainly requires more caution than we are used to, but it would be hasty to presume bad consequences for the run time operational semantics. In fact, the run time story is better than usual: as well as erasing types which play no part in computation, we can erase values which are uniquely determined.

Once we have our λ -calculus, can we encode the rest? No, thank you! We are used to Church-style encodings of data, directly representing choice of control flow, for example, encoding `Bool` so that case analysis is effectively given by the identity function

$$\text{caseBool} : \text{Bool} \rightarrow \forall P : *. P \rightarrow P \rightarrow P$$

Such an encoding can never account for an analysis which actually discovers the value, as well as choosing between branches:

$$\text{caseBool} : \forall b : \text{Bool}. \forall P : \text{Bool} \rightarrow *. P \text{ tt} \rightarrow P \text{ ff} \rightarrow P b$$

The benefit is the ability to distinguish values which require distinct treatment. The price is that we must introduce datatypes directly, doing in theory what we do in practice. I shall sketch the new Epigram approach, introducing a system of datatypes at a single stroke, together with native support for datatype-generic programming.

How do we program with these data? The familiar **case** construct makes sense if we can treat the possible patterns for an in-

dividual value, independently of other values. This cannot fail to be the case in simply typed languages, but it cannot hope to be the case with dependent types. What happens is much better! Inspecting one value can give you information about other values *for free*, guaranteed by type. E.g., when you inspect a list *indexed by length*, you find out if it is empty *and if its length is zero*.

Our old type systems were never strong enough to give 'information for free', so the possibility has not figured in language design until now. Dependent pattern matching programs show both the information we explicitly test and the information we receive in return. Patterns contain arbitrary terms, not just linear constructor forms, so a conventional rewriting semantics does not quite work. Happily, we can fix it [2]: typing ensures that these non-constructor forms always match, so they can and must be ignored operationally. If we update our treatment of pattern matching to reflect this interaction with the type system, we can make sure that compiled code does not go wrong by matching too much.

Without changing our basic conception of typed programming, we can have a language supporting an efficient run-time semantics, effective partial evaluation and a dependent type system. The distinctions between term and type, dynamic and static, explicit and implicit, are no longer aligned as in the Hindley-Milner system, but they have not disappeared altogether.

Having established this relatively ordinary programming language, I hope to persuade you not to program in it. Rather, you should get your computer to program in it for you, and work in a much higher-level language where types have a much stronger part to play. Our ordinary language is Epigram's *core* language, explicit enough to be executable without typechecking. Epigram source code, by contrast, is more compact, more flexible, and relies on typechecking to deliver its translation to the core. The presence of types enables a great deal of program inference!

This approach is not at all peculiar to Epigram. Advances in overloading mechanisms (especially Haskell's type classes) and generic programming make types work ever harder in high-level languages to drive the construction of code in low-level languages. The opportunity which presents itself is to take the reflective power of dependent types as the *basis* for programming language design, rather than a stroke of good luck. Crucially, the idea that types should act solely as discriminators of good programs from bad is a chain from which we are now free.

References

- [1] T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. In preparation, <http://www.e-pig.org/downloads/ydtm.pdf>, 2006.
- [2] H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.
- [3] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Types in Language Design and Implementation '07 16 January 2007, Nice, France.
Copyright © 2007 ACM supplied by printer...\$5.00